Statistical Techniques in Robotics (16-831, S21) Lecture #11 (Wednesday, March 10)

AdaBoost, Bandits (Explore-Exploit)

Lecturer: Kris Kitani *Scribes: Abhinav Agarwalla, Kshitij Goel*

1 Review

In the previous lecture, we learnt one algorithm for online supervised learning called Online Support Vector Machine (Online SVM). We review two variants of this algorithm in this section (Hard- and Soft-SVM).

1.1 Max-Margin Classifier

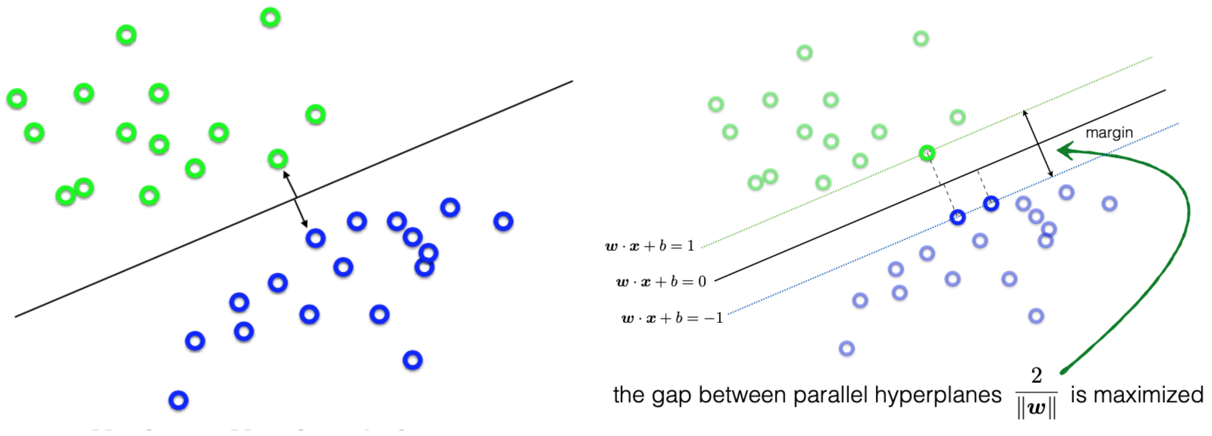
margin
$\boldsymbol{w} \cdot \boldsymbol{x} + b = 1$
$\boldsymbol{w} \cdot \boldsymbol{x} + b = 0$
$\boldsymbol{w} \cdot \boldsymbol{x} + b = -1$
the gap between parallel hyperplanes $\frac{2}{\|\boldsymbol{w}\|}$ is maximized

Figure 1: Max-margin classification task is to find a hyperplane that separates the closest points in two classes by the maximum amount of distance. Such classification is robust to minor perturbations in data.

The max-margin classification task (1) aims at finding a hyperplane that satisfies the objective:

$$\min_{\boldsymbol{w}} \|\boldsymbol{w}\|^2 \quad \text{such that, } y_i(\boldsymbol{w} \cdot \boldsymbol{x}_i + b) \geq 1 \ \forall i = \{1, \ldots, N\}$$

However, this formulation assumes that the labeled data is linearly separable, which is a strong assumption for real-world data ("Hard-SVM"). This assumption is relaxed in the Soft-SVM.

1.2 Soft-SVM

Soft-SVM is more robust to noisy boundaries (for example, imagine in Fig. 1, if a blue label is on the other side of the margin the Hard-SVM will not provide a suitable solution). The objective for

Soft-SVM introduces slack variable ξ_i for sample i and accounts for miss-classifications:

$$\min_{\boldsymbol{w},\xi} \|\boldsymbol{w}\|^2 + C\sum_i \xi_i \quad \text{such that, } y_i(\boldsymbol{w}\cdot\boldsymbol{x}_i + b) \geq 1 - \xi_i \ \forall i = \{1,\ldots,N\}$$

Here, C is a regularization parameter. When C is small, we ignore the constraints more and hence the margin of the resulting solution is large. As $C \to \infty$, Soft-SVM tends to Hard-SVM.

1.3 Hinge Loss and Sub-Gradients

To solve the Soft-SVM, we rewrite the objective as:

$$\min_{\boldsymbol{w}} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{M}\sum_{m=1}^{M} 1 - y_m \boldsymbol{w}^\top \boldsymbol{x}_m$$

Notice that the second term is not lower-bounded, it can be a very large negative value for some correctly classified (easy) points. So, we introduce hinge loss to only consider slightly correct, slightly mistaken, or very mistaken classifications:

$$\min_{\boldsymbol{w}} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{M}\sum_{m=1}^{M} \max\{0, 1 - y_m \boldsymbol{w}^\top \boldsymbol{x}_m\}$$

But the problem with hinge loss is that it is not differentiable, so we cannot take the gradient in the usual way and use our online mirror descent machinery to find optimal weights. We instead use sub-gradients (Online Sub-Gradient Descent). There can be many sub-gradients for a convex function ("differential set"). One of the sub-gradients for the hinge loss is:

$$\boldsymbol{z}_m = \begin{cases} \mathbf{0} & \text{if } y_m \boldsymbol{w}^\top \boldsymbol{x}_m \geq 1 \\ -y_m \boldsymbol{x}_m & \text{otherwise} \end{cases}$$

Overall, SVM similar to perceptrons we saw earlier but with the addition of margin.

Algorithm 1 Soft-SVM

```
1: $\boldsymbol{\theta}^{(1)} \leftarrow \mathbf{0} \in \mathbb{R}^N$
2: for $t = 1, \cdots, T$ do
3:     $y_d, \boldsymbol{x}_d \sim \boldsymbol{D}$
4:     $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} + y_d \boldsymbol{x}_d \cdot \mathbf{1}[y_d(\boldsymbol{w}^{(t)} \cdot \boldsymbol{x}_d) < 1]$
5:     $\boldsymbol{w}^{(t+1)} \leftarrow \frac{1}{\lambda(t+1)}\boldsymbol{\theta}^{(t)}$
6: end for
7: $\bar{\boldsymbol{w}} = \frac{1}{T}\sum_t \boldsymbol{w}^t$
```

Algorithm 2 Perceptron

```
1: $\mathbf{w}^{(1)} \leftarrow \mathbf{0} \in \mathbb{R}^N$
2: for $t = 1, \cdots, T$ do
3:     RECEIVE $(\boldsymbol{x}^{(t)}, y^{(t)})$
4:     $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t)} + y^{(t)}\boldsymbol{x}^{(t)} \cdot \mathbf{1}[y^{(t)}(\boldsymbol{w}^{(t)} \cdot \boldsymbol{x}_d) < 0]$
5:     $\boldsymbol{w}^{(t+1)} = \boldsymbol{\theta}^{(t)}$
6: end for
```

2 Summary

Finding the solution to the Soft-SVM objective was based on using online sub-gradient descent. Now we study the AdaBoost algorithm which uses randomized weighted majority for its solution.

Before we get into describing the algorithm in detail, we have to define a theoretical framework to judge a *weak* versus a *strong* learning algorithm. This framework is called the Probably Approximately Correct (PAC) learning model introduced in [5].

2.1 PAC Learning Model

The PAC learning model is a theoretical framework to answer two questions:

1. What is the optimal dataset size to obtain good generalization?
2. What is the computational cost of learning?

Consider an unknown distribution $P(x, y)$ from which a dataset $\mathcal{D}$ of size N is drawn (i.e. the class labels are determined by some unknown deterministic distribution $y = f^*(x)$). The dataset $\mathcal{D}$ can be viewed as an approximation of the original distribution $P(x, y)$. Now, let $\mathcal{F}$ denote the space of functions and $\mathcal{F} \mid \mathcal{D}$ denote the subset of $\mathcal{F}$ containing the functions defined on the basis of the dataset $\mathcal{D}$. Then, a function $f(x; \mathcal{D})$ drawn from $\mathcal{F} \mid \mathcal{D}$ can be considered a classifier learned on the dataset $\mathcal{D}$ and thus an approximation to the deterministic labeling function $f^*(x)$.

Definition 1. Generalization The function $f(x)$ is said to have a good generalization if the expected error rate is below a pre-defined threshold ϵ, i.e.,

$$\mathbb{E}_{P(x,y)}[\mathbf{1}[f(x; \mathcal{D}) \neq y]] < \epsilon.$$

Definition 2. PAC Learning Algorithm A learning algorithm that requires **Definition 1** to hold with a pre-defined probability $1 - \delta$ for any dataset $\mathcal{D}$ drawn from $P(x, y)$.

Definition 3. Strong PAC Learner A PAC learning algorithm whose output $f(x)$ produces an error of at most ϵ with a probability $1 - \delta$.

Definition 4. Weak PAC Learner A PAC learning algorithm whose output $f(x)$ produces an error of at most $\epsilon \geq 1/2 - \gamma$ with a probability $1 - \delta$, for some $\gamma > 0$.

At a high-level, the weak PAC learner is flexible in terms of the maximum error while its strong counterpart is not and requires a maximum error of ϵ. Further, there is a well-known result that relates the two types of learners via boosting, as shown in [4].

2.2 AdaBoost

The theorem based on [4] suggests that a weak PAC learner can be called multiple times to generate a sequence of hypothesis with a different subset of $\mathbf{D}$ which can be combined to form a single strong PAC learner. This process is called *boosting.*

Adaboost [2] is a specific learning algorithm in the class of boosting algorithms. In previous online learning methods, one receives new data points at every time instant. In Adaboost, we receive *weak learners* instead of data points. The final prediction rule is a weighted sum of learner predictions upto the current time T. Since we have T weak learner at time step T, we just use the 'number of weak learners' terminology over time steps.

We study the prediction and update step for Adaboost in the next section. The Adaboost algorithm is detailed in Algorithm 3.

2.2.1 Prediction Step

Adaboost assumes access to a fixed dataset $\mathbf{D}$, over which it defines sample-based weights $\mathbf{w}$. $\mathbf{w}_i^t$ denotes the weight for i^{th} sample $\mathbf{x}_i$ and t^{th} learner $h^{(t)}$. It's important to not that weights $\mathbf{w}$ are now indexed over samples rather than features/experts. The sample weights $\mathbf{w}^{(0)}$ are initialised uniformly. At each prediction step, a weak learners learn a classifier based on a weights loss using normalised sample weights.

2.2.2 Update Step

At each update step, the mean error $\epsilon^{(t)}$ associated with the learned hypothesis $h^{(t)}$ is computed. The error computed is again a weighted accuracy, weighed by the sample weights. Using $\epsilon^{(t)}$ and penalty parameter $\beta^{(t)}$, the weights $\mathbf{w}^t$ are updated for next learner. For a good classifier $\beta^{(t)} \to 0$, while for a bad classifier $\beta^{(t)} \to \infty$.

Interestingly, the update equation for Adaboost is exponential. This comes from the observation that weights $\mathbf{w}$ are normalised to act like a probability over samples and enforces *entropic regularisation* over the weights. We know that Online Mirror Descent for a linear objective function and entropic regularisation has an exponential update equation.

Now, let's see if the updates make sense. If the weak leaner $h^{(t)}$ classifies a data point correctly $|h^{(t)}(\mathbf{x}_n^{(t)}) - y_n^{(t)}| \to 0$, $\beta^{1-|h^{(t)}(\mathbf{x}_n^{(t)})-y_n^{(t)}|} \sim \beta$. If it doesn't, the update is $\sim \beta^2$. In other words, the weights for mis-classified samples are increased while decreased for correctly sampled. This makes sense, since the next weak learner $h^{(t+1)}$ would have a higher penalty on misclassifying the incorrect samples.

Once all the weaker learner hypothesis' are learned, the final prediction is weighted average of all weak learner. The weights is inversely proportional to the error of weak learners, with higher weights for a learner that makes fewer mistakes.

The error bound for the final hypothesis $\epsilon \leq 2^T \prod_{t=1}^{T} \sqrt{\epsilon_t(1-\epsilon_t)}$.

Algorithm 3 Adaboost

```
1: Input: Dataset $\mathbf{D} = \{\mathbf{x}_n, y_n\}_{n=1}^N$, where feature vector $\mathbf{x}_n \in \mathbb{R}^M$ and label $y \in \{0,1\}$
2: Input: Weight vector $\{w^{(0)}\}_{n=1}^N$ initialised uniformly
3: Input: Number of weak learners $T$
4: for $t = 1, \cdots, T$ do
5:     $\mathbf{p}^{(t)} = \mathbf{w}^{(t-1)}/(\sum_n w_n^{(t-1)})$
6:     $h^{(t)} = \texttt{WEAKLEARNER}(\mathbf{D}, \mathbf{p}^{(t)})$                ▷ Prediction Step
7:     $\epsilon^{(t)} = \sum_n p_n^t |h^{(t)}(\mathbf{x}_n) - y_n|$
8:     $\beta^{(t)} = \frac{\epsilon^{(t)}}{1-\epsilon^{(t)}}$
9:     $w_n^{(t)} = w_n^{(t-1)} \beta^{1-|h^{(t)}(\mathbf{x}_n^{(t)} - y_n^{(t)}|}$   $\forall n$          ▷ Weight Update Step
10: end for
11: $h_F(\mathbf{x}) = 1\left[\sum_{t=1}^{T}(log(\frac{1}{\beta^{(t)}}))h^{(t)}(\mathbf{x}) \geq \frac{1}{2}\sum_{t=1}^{T} log(\frac{1}{\beta^{(t)}})\right]$
```

2.2.3 Adaboost and Weighted Majority Algorithm

Differences. The major difference between Adaboost and WMA is that Adaboost weights samples instead of experts or features as done by Weighted Majority Agorithm (WMA). Adaboost learns a new weak hypothesis at each prediction step, while WMA just selects a hypothesis/expert randomly.

Similarities: Both utilise entropic regularisation over weights, and hence have exponential updates.

2.3 Multi-Armed Bandit

We now shift from online supervised learning to studying Multi-Armed Bandit (MAB) problem.

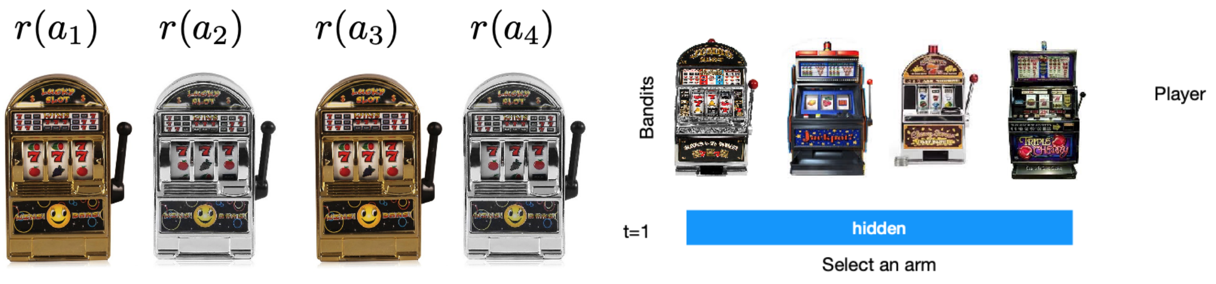

Figure 2: Introduction to the Multi-Armed Bandit (MAB) problem. (Left) The problem assumes an unknown reward distribution behind multiple slot machines ("bandits"). (Right) When a player pulls an arm and gets an output ("reward"), it is not shown the outputs ("rewards") of other arms (partial observability). This is in contrast to the Prediction with Expert Advice (PWEA) scenario where the player knows the outputs of all the experts (full observability).

Figure 2 illustrates the multi-armed bandit problem. A few observations about the feedback in this learning problem:

1. **One-Shot Feedback**: This is because one action from the player leads to one reward and selecting any particular action does not change the state at the next time-step.
2. **Exhaustive Feedback**: The player can pull all the arms for the duration of the game. The state and action spaces are finite.
3. **Evaluative Feedback**: The player receives a reward sampled from the underlying unknown reward distributions at each time-step.

Based on this type of feedback, the goal is to maximize the total reward the player receives over a horizon. First of all, since we are thinking about "rewards" instead of "loss", the definition of the regret for the learning algorithm becomes:

$$R^{(T)}(h) = \sum_{t=1}^{T} g(h(\boldsymbol{x}^{(t)}), y^{(t)}) - \sum_{t=1}^{T} g(\hat{y}^{(t)}, y^{(t)})$$

where $g^{(t)} = 1 - \text{loss}^{(t)}$ is the reward function.

Applications of the MAB problem include: A/B testing (figuring out which advertisement to display based on how the user interacts with the advertisement experience), robotic grasping (which way should the robot pick up an object), medical treatment (treating patient's health status as reward, advise treatment).

2.3.1 Exploration-Exploitation Trade-off

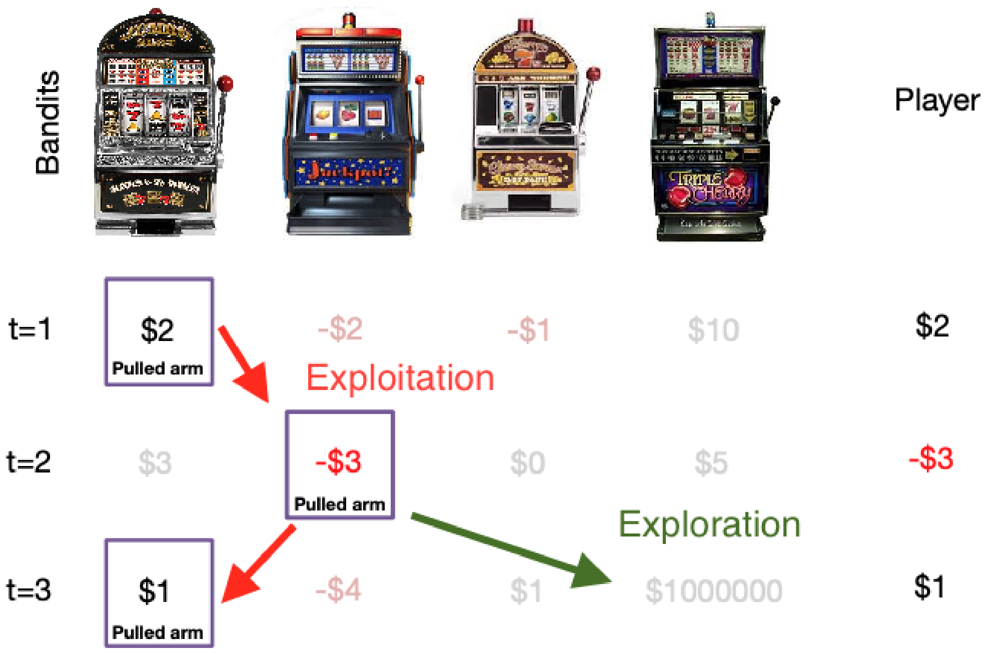

Figure 3: Illustration of the Exploration-Exploitation trade-off.

Since there is only a partial observability in the MAB scenario, we only know about the performance of a subset of the arms in hindsight. Thus we have two choices:

1. **Exploitation**: Player picks the arms that they know have performed well in the past.
2. **Exploration**: Player tries picking up new arms for which they have zero evidence.

Figure 3 illustrates this point. The player only follows the exploitation strategy and misses out on a high reward. If the player had followed a *balanced* strategy that allowed for a *trade-off* between exploration and exploitation, then it might have gotten a larger reward.

A general strategy to avoid this situation is to design algorithms that transition from exploring to exploiting. Next, we look at one such strategy in a *stochastic* environment, where each arm has a static reward distribution so that each pull gives a sample from the underlying static distribution. This is precisely the notion of a *stochastic bandit* presented in [1].

2.3.2 Explore-exploit algorithm

Assuming a stochastic bandit, how should the player pull the arms if they have only T rounds? The Explore-Exploit algorithm aims at answering this question. At a high-level, the algorithm proceeds in two phases:

1. **Explore Phase**: Pull each arm M times to estimate the mean reward.
2. **Exploit Phase**: Keep pulling the arm with the highest expected mean reward until T.

Note that there is an assumption here that the total number of trials T is greater than or equal to the product of the number of arms and M.

More formally, let $a_k \in \mathcal{A}$ denote the action for a particular arm and the cardinality of the set $|\mathcal{A}| = K$ represent the total number of arms. The reward received at time t is denoted by $r^{(t)}$, the number of exploration steps (per action) by M, and the estimated average reward of action k by $\hat{\mu}_k$. The algorithm is given by:

Algorithm 4 Explore-Exploit

```
1:  for $k = 1 \to K$ do
2:      for $m = 1 \to M$ do
3:          $a = k$
4:          Receive($r$)
5:          $\hat{\mu}_k = \hat{\mu}_k + \frac{r}{M}$
6:      end for
7:  end for
8:  for $t = KM \to T$ do
9:      $a^{(t)} = \arg\max_{k'} \hat{\mu}_{k'}$
10:     Receive($r^{(t)}$)
11: end for
```

Lines 1 to 7 show the exploration phase where each arm k is tried for a total of M times to estimate the mean reward of the underlying static reward distribution for each arm. Afterwards, lines 8 to 11 show the exploitation phase which continues after KM timesteps that the exploration phase took until the end at T. Next, we look at the regret bound for this algorithm.

2.3.3 Regret Bound for Explore-Exploit

The regret bound for the Explore-Exploit algorithm is given by $O(K^{1/3}T^{2/3})$. The derivation of this bound requires us to know about the Hoeffding's Inequality [3].

Theorem 5. Hoeffding's Inequality Consider a one-dimension distribution ν with expectation μ, where any sample $r \sim \nu$ is bounded such that $r \in [0, 1]$. Given T i.i.d samples, $\{r^{(t)}\}_{t=1}^{T}$, we have that for any ϵ:

$$p\left(\left|\sum_{t=1}^{T} \frac{r^{(t)}}{T} - \mu\right| \geq \epsilon\right) \leq 2e^{-2T\epsilon^2}$$

Intuitively, this inequality means that the estimate of the mean gets better with the more samples are available.

References

[1] H. Chernoff. Sequential design of experiments. *The Annals of Mathematical Statistics*, 30(3):755–770, 1959.

[2] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[3] W. Hoeffding. Probability inequalities for sums of bounded random variables. In *The Collected Works of Wassily Hoeffding*, pages 409–426. Springer, 1994.

[4] R. E. Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[5] L. G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, 1984.